Received: 30 September 2017 | Revised: 28 February 2018 | Accepted: 5 March 2018
DOI: 10.1002/cpe.4494

WILEY

SPECIAL ISSUE PAPER

A mirroring architecture for sophisticated mobile games using computation-offloading

M. H. Jiang[1,2] | Otto W. Visser[2] | I. S. W. B. Prasetya[1] | Alexandru Iosup[2]

[1]Utrecht University, Utrecht, the Netherlands
[2]Delft University of Technology, Delft, the Netherlands

Correspondence
I. S. W. B. Prasetya, Utrecht University, the Netherlands.
Email: s.w.b.prasetya@uu.nl

Funding information
NWO/STW Vidi grant MagnaData, Grant/Award Number: 14826; NWO/STW Veni grant AtLarge, Grant/Award Number: 11881; Dutch projects COMMIT and Commissioner

Summary

Mobile gaming is already a popular and lucrative market. However, the low performance and reduced power capacity of mobile devices severely limit the complexity of mobile games and the duration of their game sessions. To mitigate these issues, in this article, we explore using computation-offloading, that is, allowing the compute-intensive parts of mobile games to execute on remote infrastructure. Computation-offloading raises the combined challenge of addressing the trade-offs between performance and power-consumption while also keeping the game playable. We propose *Mirror*, a system for computation-offloading that supports the demanding performance requirements of sophisticated mobile games. *Mirror* proposes several conceptual contributions: support for fine-grained partitioning, both offline (set by developers) and dynamic (policy-based), and real-time asynchronous offloading and user-input synchronization protocols that enable *Mirror*-based systems to bound the delays introduced by offloading and thus to achieve adequate performance. *Mirror* is compatible with all games that are tick-based and user-input deterministic. We implement a real-world prototype of *Mirror* and apply it to the real-world, complex, popular game OpenTTD. The experimental results show that, in comparison with the non-offloaded OpenTTD, *Mirror*-ed OpenTTD can significantly improve performance and power consumption while also delivering smooth gameplay. As a trade-off, *Mirror* introduces acceptable delay on user inputs.

KEYWORDS

cloud offloading, computation-offloading, fine-grained offloading, mirrored computation, offloading mobile games, offloading policies, system architecture

1 | INTRODUCTION

The mobile gaming market offers services to over 2 billion gamers, generating yearly over $100 billion and over 40% of the lucrative gaming industry.[1] However, mobile devices such as smart phones and tablets have limited computational power and power supply, which severely limit the complexity of mobile games and the scale of online gaming sessions. The battery life can be depleted very quickly by a combination of, for example, heavy computations or network and camera usage,[2] which indicates that traditional sophisticated games cannot yet run long on mobile platforms. With *sophisticated* games, we mean games that have complex gameplay mechanics. These games can often have a high number of game objects in the game world at the same time with many possible interactions between these objects and the user. This makes these games a challenge to run smoothly on resource-constrained mobile devices and to be able to play them without rapidly depleting the device's battery. A potential approach to mitigate these problems is through *computation-offloading*, that is, letting remote machines run the compute-intensive parts of mobile games and letting the mobile clients only receive the results. Computation-offloading promises to free up local resources on the mobile client and possibly save much of its valuable battery power.[3] However, offloading also raises new challenges. Offloading requires more complex software engineering, it can introduce latency that slows down rather than speeds up the game, consume rather than spare power, and cost extra overall resources to maintain the offloading infrastructure. This is confirmed by earlier approaches to offloading,[4-6] which so far have led to insufficient performance for the requirements of sophisticated modern games[6-8] or only work for simple games.[9-11] Addressing these challenges is crucial if we are to actually profit from offloading, which is the goal of this work.

Deciding what to offload from a mobile game is an important key to address this challenge. A large body of offloading work has emerged in the last decade, focusing on different approaches targeting specific applications.[12,13] The most beneficial and easiest functions to offload are *coarse-grained functions*, that is, functions that are resource intensive, have small-sized input and output (thus minimizing communication overhead), and can tolerate a long response delay.[3] The benefits of offloading these kinds of functions for mobile games have indeed been proven.[4,9-11] The effects on the client-side for all these offloading frameworks are all fairly similar: they improve performance, lower local resource usage, and decrease power consumption. However, these frameworks only work for simple games with mostly coarse-grained functions and will not work well for *fine-grained functions*, which are functions that are not very computationally intensive on their own but may still contribute a significant portion of the computational load of the game due to the high frequency at which they are being called. Frameworks in previous works all process each call to an offloadable function individually, creating high overhead and waiting time for network communications each time a function needs to be offloaded and thus making the game unplayable when offloading fine-grained functions.

This article presents *Mirror*, the first computation-offloading architecture suitable for sophisticated mobile games (Section 3.1). *Mirror* is compatible with all games that simulate the game world in ticks and are user-input deterministic. Both of these concepts are further explained in Section 2. The architecture of *Mirror* consists of several inter-related controllers, mirrored across client and offloading-server systems. Its simplicity hides several key contributions to the field.

First, *Mirror* enables offline and online program partitioning with both fine- and coarse-granularity (Section 3.2). Game developers specify at the source code level which computation is to be offloaded (offloadable entities), and at runtime, *Mirror* dynamically re-partitions these entities between clients and servers.

Second, *Mirror* takes and controls dynamic offloading decisions using configurable offloading policies operating on offloadable entities of any granularity (Section 3.3). This enables a simple yet powerful mechanism where both game developers and game players can express their needs, enabling, for example, games where high performance is enabled while connected over a local Wi-FI, whereas low bandiwdth consumption could be preferred while on roaming charges or low power consumption during periods away from charging opportunities.

Third, *Mirror* also defines two protocols for synchronization between client and offloading-server. In contrast to *lockstep synchronization*, which is the *de facto* standard for building military simulations[14] and RTS games,[15] *Mirror* considers an asynchronous protocol where the server simulates the game ahead of the client, with the simulation horizon automatically under control in the running system. Doing this allows *Mirror* to avoid the limitations of synchronization but requires an additional protocol to cope with user input, which introduces uncertainty in the game (Section 3.5). Combined, the two protocols enable *Mirror*-based systems to bound the delays introduced by offloading. A more desirable trade-off between computational performance increase and game playability is thus achievable.

To gain insight on *Mirror*'s actual performance, we have implemented a prototype of the system and conducted extensive real-world experiments (Section 4). A highlight of our experiments is that we have adapted a real-world sophisticated online game to use *Mirror*, equipped with a library of policies that offer interesting performance-bandwidth-power trade-offs. Our thorough examination of different scenarios reveals effects, both positive and negative, of offloading the computational load of this game. In six carefully crafted experiments, we record, for this purpose performance, network consumption and power consumption data but also in-game delays and stutterings caused by offloading. We analyze these results (Section 5) and show strong evidence of the usefulness of computation-offloading for sophisticated mobile games and of how the policies we have designed are able to manage the trade-offs. Last, we identify promising directions for future research and experiments but also analyze for this work the known limitations and threats to validity (Section 6).

Our work stands out from the large body of related work in Section 7. A summary of Mirror was presented as a short-paper.[16] With respect to this earlier publication, the current article adds much. It details the *Mirror* architecture, and, in particular, it introduces its core components. It presents program partitioning in much greater detail, and in particular details the offloadable entity interface; this is key to showcasing the limited engineering work required to use *Mirror*. *Mirror*'s dynamic offloading and the two protocols, which are key parts of our contribution in this work, were not presented in our previous work[16] and are now presented here. This article explains the complex setup used to conduct real-world experiments and introduces new results on the impact of dynamic offloading policies, on the use of local and cloud-based infrastructure to offload to and on *Mirror*'s impact on bandwidth usage.

The remainder of this article is structured as follows. Section 2 refines a set of requirements for supporting sophisticated mobile games through offloading. Section 3 explains how the *Mirror* framework works. Section 4 describes the experiments we conducted to examine the framework, and Section 5 presents the results of these experiments. Section 7 surveys the related work, and Section 6 addresses the limitations and opportunities for the future of this work. Finally, Section 8 concludes.

2 | REQUIREMENTS FOR COMPUTATION-OFFLOADING IN SOPHISTICATED MOBILE GAMES

We formulate in this section the main requirements for a computation-offloading system targeting *sophisticated mobile games*, that is, mobile games that are complex, real-time, and used in the real-world (R1). The concrete requirements are summarized at the end of this section.

The basic goal of this work is a system able to run a game whose computational load is partitioned across a game client and an offloading server, which is essentially running an additional instance of the game. Because mobile devices encounter periods of network unavailability, the game should still be playable without offloading[17] (part of main requirement R1, in the end-section summary). When a part of the computational workload is to be offloaded (R2), the client skips the computation and instead relies on results computed and then sent back by the server. We aim to support games whose computational tasks are fine-grained (R3), perhaps no larger than a single in-game object, eg, a moving in-game vehicle. The offloading system must decide what gets offloaded, when, and how. We aim to provide control over how offloading decisions are made to both game developers and game users (R4).

Some form of synchronization is needed between the client and the offloading-server (R5). Because synchronization in general is a complex problem without a single general solution, we focus in particular on games whose in-game world is updated in discrete steps (R1.a). These steps ("*game ticks*") advance the in-game time (simulation time) with a fixed increment. The in-game time can thus be denoted as the tick the game is currently at. The real-time duration of a tick may however varies; real-world games do this to control how the player perceives the progress of the in-game time. "*Tick-based*" is the *de facto* standard in the industry for (real-time) simulation and strategy (RTS) games and is also commonly used in many virtual reality and simulation environments in military, medical, and other application-domains.

The games we aim to support must also be "*user-input deterministic*" (R1.b): if the game is run on the same initial state and is given the same sequence of user inputs, scheduled in the same way (in terms of in which ticks they occur), the resulting state after each tick must be exactly the same no matter under which conditions the game is run. To make games take decisions that appear to be random to the player, the *de facto* standard in the industry is to use pseudo random-number generators.

In summary, the main requirements are as follows; we aim for a system that:

- **(R1)** supports sophisticated mobile games that are (a) *tick-based* and (b) *user-input deterministic*. The game should continue to work even during (temporary) lack of network access;
- **(R2)** offloads compute-intensive tasks;
- **(R3)** able to do fine-grained offloading;
- **(R4)** allows developer- and user-control on the offloading operations;
- **(R5)** synchronizes clients with the offloading-server.

3 | THE MIRROR ARCHITECTURE

We present in this section the *Mirror* architecture for computation-offloading targeting sophisticated mobile games. By design, *Mirror* defines the roles and operation of the client and of the offloading-server (from hereon, *server*); the server is running on a more capable machine, processing any *part* of game-related computation faster than the client could. This architecture is designed to meet the requirements R1–R5: offloading (R2 and R3) and synchronization (R5) are completely managed by the *Mirror* architecture, at development- and at run-time, even during (temporary) network unavailability (R1). The client decides which computation to offload based on offloading policies (R4), whereas the server always performs the complete computation. Sections 4 and 5 show empirically that the *Mirror* architecture can be implemented in practice for a real-world sophisticated game (R1), for several mobile hardware platforms and for several cloud-based offloading infrastructure.

3.1 | Overview of the Mirror architecture

The core operational principle of *Mirror* is to have the client and the server *mirror* each other's operation. The architecture, which we depict in Figure 1, consists of only a handful of components: the *Mirror* Framework Controller, the Game Offloading Controller, the Offloadable Entity (described in Section 3.2), and the (dynamic) Offloading Decision Controller running policy-based decisions (Section 3.3). Except for the latter component, components are mirrored from client to server. The *Mirror* architecture also consists of protocols for real-time asynchronous offloading from client to server (Section 3.4) and for synchronization between client and server (Section 3.5).

The *Mirror Framework Controller* component is responsible for managing the whole offloading process (requirement R2), including the communication between the client and the server. To do this, this controller needs some degree of control over the game-loop of the game itself, which it acquires through the use of the *Game Offloading Controller* interface. The developers of the game must implement the functions in this interface, which consists of only basic operations: starting and ending a game, pausing and resuming the game-loop, and retrieving the current in-game time. To run a game, the client first connects to the offloading server to establish an *offloading session*. To set up such a session, the client sends a saved-game to the server or instructs the server to load a prepared saved-game to be used as the the game *starting point*. The game can now start, with both instances of the game running but the server doing much of the computation on behalf of the client.

Elements of the game whose computation is (partially or completely) offloadable must implement the *Offloadable Entity* interface; the latter contains *hooks*, that is, predefined asynchronous mechanisms for remote function-calling used by the architecture to partition computation between client and server. Not every offloadable computation is actually offloaded; this is decided dynamically by the *Offloading Decision Controller* component, client-side, and the decisions are communicated to the server. If a computation C is offloaded, the client skips its execution and expects the results from the server. The server always run the full execution of the game, including the execution of C. When C finishes, the server will send the result to the client, along with the in-game time t at which the result takes effect. Upon receiving this result and upon arriving at the in-game time t,

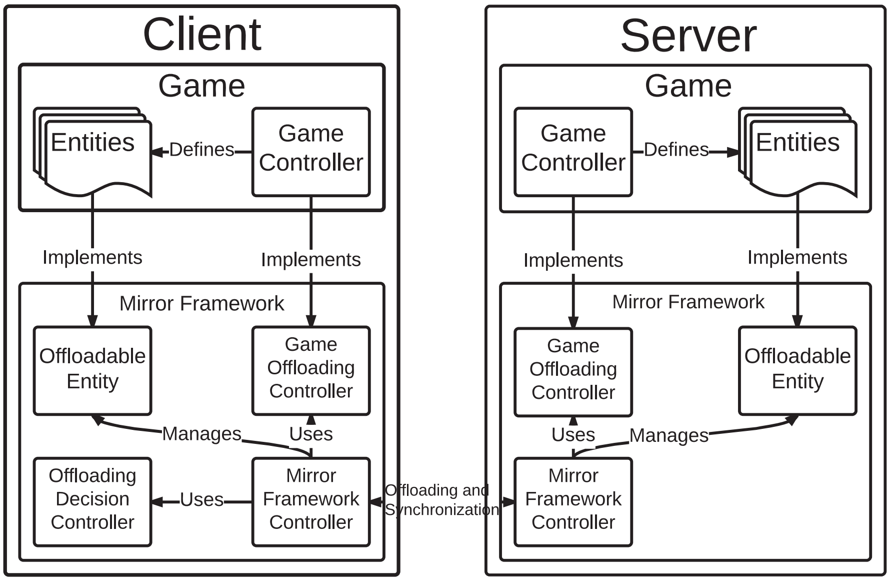

FIGURE 1 *The architecture of Mirror and its components.* The client and server both run the same game

the client will perform the actions required to integrate the result into its local game-state. Thus, the client and the server will arrive at the same game state for each game-time *t* as if the client has executed *C* itself. Importantly, this synchronization scheme does not constrain the two game instances, on the client and on the server, to run at the same speed.

Having the full game mirrored on both the client and the server indeed implies that some computation will be duplicated as opposed to simply having an online game with a strict division of client and server computation. However, the duplication gives us two important benefits. First, with *Mirror*, the user can still play the game when no Internet connection is available (part of R1). Second, because the server is not constrained to run with the same speed, the server is able to run ahead to predict and send the information that the client will need in the near future *ahead of time*! This means that the client does not need to pause its own simulation, which is essential to keep the game playable. How this is done is explained in Section 3.4.

3.2 | Offline and online program partitioning through offloadable entities

To benefit from *Mirror*, developers need to specify which parts of the game's source code represent offloadable computation in an *offline* process of *program partitioning*. Based on our experience with this process, when adapting a sophisticated mobile game to leverage *Mirror* (described in Section 4.1), the effort to do this is reduced.

Addressing requirement R3, *Mirror* leverages the fine-grained structure of sophisticated mobile-game codes. For such games, in-game objects are often programmed as object-oriented (OO) or OO-like *classes*. Each of these classes defines the in-game behavior of its objects/instances. At run-time, multiple instances of the same class can be created. In the most demanding (non-offloaded) RTS games and simulations, hundreds to thousands of such instances can coexist.[15]

In *Mirror*, developers can make an existing game-object class *O* offloadable by making it implement the *offloadable entity* (OE) interface. The OE interface provides a hook for the developers to separate the behavior of *O* into offloadable and non-offloadable parts. For each offloadable part, a function must be defined, to be executed by the server, to serialize and send the result of a call back to the client, and a function (to be executed by the client) to handle and process such a result. Figure 2 indicates, through an OE example, that specifying a part of code as offloadable does not require significant engineering work. The technical details of the Figure follow the logic of defining OEs introduced earlier in this section. The function `Update()` does two tasks: it calls `Move()` and then `DoPathPlanning()`. The function `Move()` is always executed, whereas `DoPathPlanning()` has been specified, through the OE interface, as *offloadable*. The latter means that `DoPathPlanning()` is only executed on the client if the entity is currently not being offloaded. If an instance of the class is marked as offloaded and a path-planning event is triggered on the server side, a message will be sent by the server by creating a custom payload and sending it by calling `SendEventMessage(payLoad)`. When the client receives the message, it will pass its contents to `ReceiveEventMessage(payLoad)`, which will asynchronously trigger an update of the client-side internal variable `direction`.

In the *Mirror* architecture, *offloadable instances* (OIs) are in-game instances of an OE, and in *Mirror*, each of them can be offloaded independently from all the others. Offloading a compute-part of an OI is achieved through a simple mirrored approach. When the client decides to offload an instance, it subscribes to get the results of all future calls to the offloadable parts of the instance. If the client decides to start or to stop offloading an instance, it will communicate this decision with the server, sending: (1) the identifier of the instance (OI), (2) the identifier of the offloadable part of the OI, and (3) the decision to start or stop. The server acts as a mirror: each time the server finishes a call to an offloadable part of an instance, it will send an *event message* (EM) to the client with: (1) the identifier of the instance, (2) the identifier of the function call, (3) the result of the call, and (4) the

```
Vehicle : OffloadableEntity {
  void Update() {
    Move(); // not ofloadable
    if ((IsClient & !Offloaded) || IsServer) {
       DoPathPlanning(); // offloadable
       if (IsServer && Offloaded) SendEventMessage(newDirection);
    }
  }
  void ReceiveEventMessage(payLoad) { direction = payLoad; }
}
```

FIGURE 2 *Example code of a simplified class, "Vehicle", implementing the OE interface.* See text for details

in-game time at which the call occurred. Upon receiving an EM, the client processes the message by passing the EM and calling the corresponding processing function of the OI associated with the message.

As remarked before, converting an ordinary game-object class into an OE does not take much work. This is also confirmed by our experience when converting the Open Transport Tycoon Deluxe (OpenTTD) game to make it offloadable for the case study described in Section 4. To make an existing game offloadable, developers first have to identify which classes and which code within the classes would be beneficial for offloading. This boils down to estimating the performance of a specific piece of code. Admittedly, this is not easy, but it is not necessarily very hard either. Modern games are typically very modularized, eg, by following the OO approach. Play testers can quickly point out which types of game objects would spawn a lot during typical game plays. These types would map to their corresponding OO classes as potential candidates for offloading. Developers also often know obvious loops that require a lot of computational power, eg, those loops implementing functionalities like collision detection, path planning, and AI. Then, profiling can be used to confirm the candidates or to reveal if there are more pieces of code that would make potential candidates for offloading. In our OpenTTD case study, the parts we chose to offload was simply determined by looking at the code and finding loops that look very intensive. This did not require much work at all even without prior knowledge of how the code of OpenTTD is structured. The results (Section 5) reveal that even such a basic approach is already very beneficial. However, when the code of an existing game is not very well structured or follows the OO approach poorly, it can be hard to pick out functions that can be offloaded easily. This can happen, for example, for functions that have a large number of side-effects on the gamestate. In such a case, a restructuring of the game's code might be needed to still be able to offload such a compute-intensive part. When rewriting a game's code or creating a completely new game for offloading, it is therefore important to put compute-intensive code in functions with limited scopes as much as possible.

3.3 | Dynamic offloading decisions and offloading policies

At run-time, *Mirror* has to decide on offloading OIs. Although all OIs are offloadable, they are not always offloaded. Instead, the decision to actually offload them is taken dynamically, during the offloading session. Because the server is more powerful than the (mobile) client, simply offloading all OIs could lead to maximum performance, but it may not always be desirable due to bandwidth usage and power consumption and to Internet-issues such as delays and jitter. Addressing this situation, in the *Mirror* architecture, the *Offloading Decision Controller* component decides *online* on which OIs should be offloaded, when, and for how long. The frequency of these offloading decision moments can be adjusted according to the preferences of the developer or the user. Addressing (R4), game developers can control this decision making process by specifying *offloading policies*. Several policies are introduced below; their impact on performance is analyzed in Section 5.6.

The *Offloading Decision Controller* is responsible for making offloading decisions, client-side. These decisions can be taken based on a variety of statistics about the resource usage on the client machine and based on the preferences set by the user and/or by the game designers. During an offloading session, *Mirror* will collect various metrics (see Section 4.4) such as the current in-game progress rate and delivered *frames per second (FPS)*, bandwidth usage, and number of already offloaded instances. The active offloading policy makes offloading decisions based on these data-points. Without developer annotations or profiling information, *Mirror* policies have to make guesses about which and how many instances to offload and for how long, aiming to satisfy the criteria for which they were designed. The policies achieve this using a trial-and-error approach coordinated by the *Offloading Decision Controller*, by changing the number of offloaded instances, looking at the effect of it, then proceeding to make more changes. They may thus need time to converge to a good value for the number of offloaded instances.

We equip *Mirror* with a library of four policies: (1) The *target FPS/TPS policy* monitors the current FPS of the game and attempts to offload just enough instances to maintain a user specified FPS or the equivalent policy considering *ticks per second* (*TPS*) instead of FPS; (2) the *bandwidth cap policy* offloads, similar to policy (1), as many instances as possible within the user specified limits on download and upload rates; (3) the *coarse-grained only policy* will only offload OIs that are marked as coarse-grained, using developer-provided hinting or profiling data; and (4) the *offload all policy* simply offloads all possible instances at any given time, for maximum performance.

Game developers can create new offloading policies by writing a decision-making function focusing on specific design and performance goals. Developers can also define adjustable parameters for these policies, which players of their games can use to further adjust the games at run-time. In our experience with developing the four policies already available in the *Mirror* prototype, the required design, engineering, and testing work incur reasonable overhead.

3.4 | Protocol for real-time asynchronous offloading

During an offloading session, the client and the server in *Mirror* must synchronize their execution to make sure that their game states are consistent with each other and thus addressing requirement R5. To synchronize, the *de facto* standard in the industry is to use the *lockstep synchronization* protocol, especially for military simulations[14] and RTS games.[15] For this protocol, both the client and the server advance their game ticks together. Thus, their states are always identical at the beginning and end of every tick. The client and the server wait to advance the current tick until they agree on the (offloaded) computation to occur between the current and next ticks. The server then performs the computation and sends the results to the client, and the client incorporates the results' effect. Only then will both advance to the next tick. The main drawback of this protocol is that the first OI processed for the current tick whose computation is to be offloaded cannot receive its needed result from the server until the last OI processed in the previous round finishes incorporating its offloaded result. If the game has many OIs, this scheme will slow down the game unacceptably.

To improve performance, and contributing to R5, we design for *Mirror* an asynchronous protocol where *the server runs ahead of the client*. Intuitively, this is useful because much of the computation required by games is due to regular in-game updates or a reaction to input from the user from the relatively far-past. When the client has to offload a computation, the server, who is running ahead due to executing on a more capable machine, has already sent the result, which the client can immediately pick up from its own message queue.

The protocol we propose for *Mirror* has the following steps. When the server has to execute an offloaded function, it will execute it as usual and send the result to the client as a time stamped EM. The client runs behind its offloading-server, but the 'user-input deterministic' requirement (R1.b) means the client's computation should simply be a mirror of the server. In due time, the client arrives at the same function call (OI update) that the server already did, and its state would be exactly the same as the server's state when at that tick. As mentioned in Section 3.1, the client transforms automatically the call into a skipped computation and relies on the result returned by the server instead. Ideally, at the moment of the skip, the corresponding EM should already be in the client's queue, which means the client does not need to wait every time it wants to offload a function call. The EM's time stamp indicates that it has to be executed at a scheduled tick. In due time, the EM will be executed, and the effect of the function call is thus incorporated into the game state. This benefits the smoothness of the game (defined in Section 4.4).

The operation of the *Mirror* asynchronous offloading relies on the "tick-based" requirement (R1.a, defined in Section 2). Some computation may depend on the in-game time. For example, when the game simulates a moving vehicle, the distance the vehicle travels between the current and next ticks depends on the in-game time duration between the ticks. If the computation is offloaded, the duration between ticks must be the same for both client and server, or else, the server will produce an incorrect result. For this reason, the in-game time duration between ticks must be constant, which is ensured by the "tick-based" requirement.

Despite the server running ahead of the client, it is still possible for an EM sent by the server to arrive too late at the client, eg, due to delay in the communication network. An EM is considered *late* when the client is already at an in-game time that is later that the EM's time stamp. A late EM will desynchronize the game state between the client and the server. To reduce this risk, we impose the following rules to the operation of *Mirror*. Let *lower delay* (LD) denote the amount of time the server should be ahead of the client. LD should be chosen large enough for EMs to (very likely) arrive at the client in time. The minimum value of LD is dynamically determined at run-time by periodically measuring the round-trip-time between the client and the server and dividing it by two to estimate the one-way latency. Let the *message delay* (MD) be this minimum value. In practice, a slightly higher value than MD should be chosen for LD to take into account momentary changes in latency. The user can choose this desired safety margin by choosing a number $n_{\text{LD}} \geq 1$. At run-time, the architecture will calculate LD using $n_{\text{LD}} \cdot MD$. The intial value for LD is calculated during the startup of an offloading session. Figure 3 illustrates through an example a possible configuration of LD and MD and also includes the similar concepts UD and UID, which are explained in Section 3.5. Why the margin for LD should not be chosen infinitely large is also explained in that section.

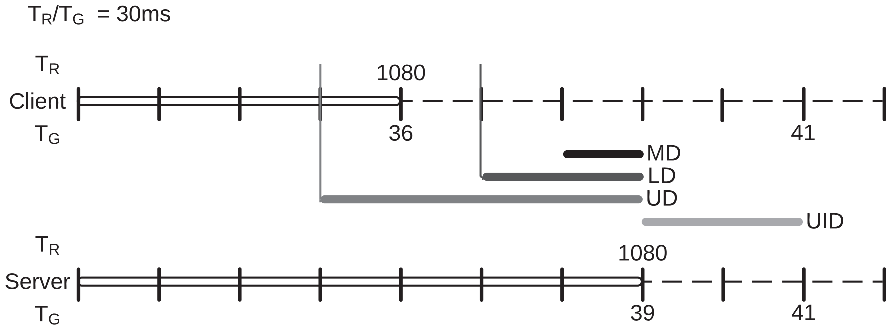

FIGURE 3 *The synchronization protocol of the Mirror architecture at work*. The progression of in-game time for the client and the server is represented as line segments. T_R is real-time in milliseconds, and T_G is in-game time in ticks. In this example situation, the game performs one tick every 30 ms. At $T_R = 1080$ ms, the client is at $T_G = 36$ and the server at $T_G = 39$. Example values for the measured MD, user chosen LD and UD, and the calculated UID are shown in ticks. The client is kept between LD and UD ticks behind the server. An EM generated at $T_G = 39$ at the server will take at least 60 ms time to arrive at the client. A user-input arriving on the client-side at $T_G = 39$ will be scheduled for execution at $T_G = 41$ for both the client and the server

During an offloading session, both the client and the server continuously notify each other of their current in-game time. The frequency at which this happens is the *update frequency* (UF). If the client detects that its own in-game time is less than or equal to LD behind the server's, the client will temporarily pause its own simulation to wait for the server. The user at the client-side can perceive these as the game stuttering. We analyze in Section 5.2 the impact of this on game smoothness in practice.

3.5 | Protocol for synchronization through user-input delay

Aside from EMs, *Mirror* also needs to synchronize user-inputs received at the client-side with the server (addressing requirement R5). Executing the effects associated with a user input immediately on the client-side will cause a desynchronization with the server, as the server not only has not been notified yet of the user input, but it also has already passed in its computation the in-game time at which the client received the input.

To solve this problem, the execution of the user input event is delayed on the client-side and rescheduled at a tick that is in the future for the in-game time of both the client *and* the server. The new tick at which the user input will be used in updates (will be executed) is calculated client-side. This is done by taking the currently known server-tick and adding to it the *user input delay* (UID), which is the number of ticks to be added to ensure the message arrives at the server on-time. UID is computed as $n \cdot \text{MD}$, where n is chosen by the user and MD has been introduced in Section 3.4. UID should account for the time it takes for the message from the client to arrive at the server and the client not having the most up to date in-game time of the server. By experimenting with different values, we discovered that a value of $n = 2$ works well with our setup (Section 4).

After the client schedules the user input and sends the message to the server, both will proceed to simulate their game as normal until they arrive at the scheduled in-game time. At that moment, they will execute the user input event.

Because the server is a faster machine than the client, the in-game time of the server will be increasingly further away from the client as real-time passes. In principle, this benefits the offloading processs described in Section 3.4, as this way, EMs will most certainly arrive on time, and the client will never need to pause its simulation to wait for the server, avoiding stuttering altogether. However, if the server is very far ahead of the client, all user input will need to be scheduled very far in the future of the client too. This will create unacceptable input delays perceived by the user, making the game unplayable. This is why LD should not be chosen infinitely large but instead should balance between game stability and smoothness on one side and user input delay on the other. (The alternative of recomputing state on the server, which means computing ahead of time is treated as speculative execution can lead to significant complications in the computational model and falls outside the scope of this work.)

The rule imposed by LD is only used to keep the in-game time of the client and of the server close. To reduce the user-input delay, it is also necessary to keep the client not too far behind the server. To do this, we define the *upper delay* (UD) and depict it in Figure 3. The value of UD is calculated in the same way as LD: by multiplying a number n_{UD} chosen by the user with the measured MD at run-time. An additional constraint is that $n_{\text{LD}} \leq n_{\text{UD}}$, whereas the rule imposed by LD is checked at the client-side, and the rule imposed by UD is checked at the server-side. If the server detects that it is more than UD ticks ahead of the client, it will pause its own simulation to wait for the client. The number of ticks calculated by UD-LD represents a buffer time for sudden network delays. A large buffer results in longer user input delays but improved game stability and smoothness under bad network conditions. A small buffer (or no buffer when $n_{\text{LD}} = n_{\text{UD}}$) results in shorter user input delays but a decrease in game stability and smoothness.

4 | EXPERIMENT SETUP

We introduce in this section the experiment setup we have used to validate and analyze the *Mirror* architecture. To conduct experiments with a *Mirror* system that implements the architecture, we develop a real-world prototype running a sophisticated mobile game. We set up the experimental environment to include accessible and relatively inexpensive devices, both on the client- and on the server-side. We further set up the infrastructure to experiment with different real-life workloads and scenarios (eg, both home computers and cloud computing settings), from which we collect diverse results (eg, metrics for performance, in-game experience, power consumption, and network consumption). The use of this setup for the specific experiments conducted in this work is explored in Section 5 and summarized in Table 2.

4.1 | A Mirror prototype

Addressing requirement R1, we have implemented a real-world prototype of the *Mirror* architecture and applied it to the complex real-world game *Open Transport Tycoon Deluxe* (OpenTTD). OpenTTD is a popular and active open-source re-implementation of the best-selling real-time strategy game *Transport Tycoon Deluxe* (1994). OpenTTD provides many new extensions over the original and allows more players and more objects to be simultaneously in the game, which makes OpenTTD much more computation-intensive than the 1994-original and a good real-world sophisticated game to experiment with. The work of Shen et al[18] discusses why OpenTTD is a suitable game for research purposes. Figure 4 depicts a typical view from this game.

In OpenTTD, players manage a transport company. The goal of the game is to earn more money than other companies by building roads and vehicles and transporting various types of goods across the map. OpenTTD is a computation-intensive game due to the high number of objects that can be in the game at the same time that must be continuously updated. The version of OpenTTD used in the experiments has a maximum of 33 TPS when running the game at normal speed (so, one tick every 30 ms); it will run at lower TPS if the machine is not able to simulate the game that fast.

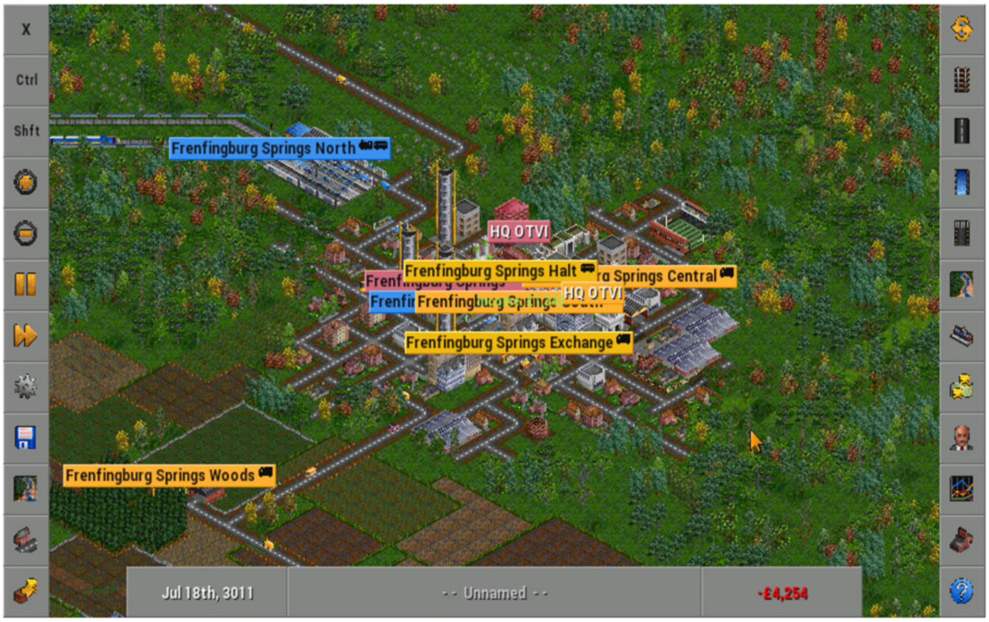
X
Ctrl
Shft
Frenfingburg Springs North
HQ OTVI
Frenfingburg Springs Halt
ra Springs Central
HQ OTVI
Frenfingburg Springs
Frenfingburg Springs Exchange
Frenfingburg Springs Woods
Jul 18th, 3011
-- Unnamed --
-£4,254

FIGURE 4 *OpenTTD as used in the experiments running on a mobile device*

The official OpenTTD is free open-source software written in C++ and developed using modern software engineering practices.* For the mobile version of the game, we use as codebase the OpenTTD Android-port,[19] version `1.4.4.36`. This version of OpenTTD can be built for both Android and Linux using the same code, which is useful for our experiments. It also allows playing in the same game with clients compiled from the core OpenTTD code and running on any platform (eg, also Windows and Mac operating systems).

Our implementation of *Mirror* offloads *2 types of objects* of OpenTTD. First, all types of road vehicles can be offloaded. The offloadable parts consist of the path-planning and the collision-detection functions. Second, the AI (artificial intelligence) of non-player–controlled companies can be offloaded. In terms of offloading granularity, the computation associated with each individual road-vehicle is fine-grained as these game-objects occur frequently but are not computationally intensive on their own. In contrast, the computation for AI decisions is much more intensive and infrequent, and thus, AI offloading is *coarse-grained*.

4.2 | Infrastructure for experiments

We have conducted experiments with OpenTTD and *Mirror*, with the client deployed on different mobile devices (2 types), and the server deployed on different infrastructure (3 types, including cloud-based infrastructure). For all devices, we focus conservatively on devices with low or moderate performance, thus representing the common devices found in possession of players or accessible to them for a reasonable budget.

As mobile devices, we used a low-performance device, Nexus 3 (Galaxy Nexus), and a moderate-performance device, Nexus 6. As infrastructure for the server, we have used a moderate-performance Samsung Q330 laptop locally, and a low-performance Amazon EC2 t2.micro, and a moderate-performance t2.medium cloud (virtual) machines located in Frankfurt. The Samsung Q330 laptop was connected to the client-device using a local LAN connection, whereas the connection between the client and the Amazon EC2 servers is WAN Internet-based. Although it is interesting to see the effects of different network technologies (like 3G and 4G) on *Mirror*, it is not the focus of this work.

4.3 | Input workloads for experiments

As *workload*, we use OpenTTD to generate gameplay workload with average computational load, using players controlled each by an artificial intelligence to play the game and gradually populate the game world with different types of entities. The parameters for the in-game map such as flatness, density of cities and industrial buildings, etc, were chosen to make playing and building on the map as easy as possible. The work of Shen et al[18] discusses how specific map settings affect the computational load of OpenTTD.

To prevent warm-up effects in our experiments, we first run OpenTTD and save the game after the in-game world is populated with load-generating objects. Because OpenTTD is tick-based (R1.a in Section 2), the save-games can act as *starting points*, that is, they can be continued in our experiments. We have created 6 starting points for our experiments, with the characteristics summarized in Table 1.

Our experiments rely on AI players instead of real-user interactions. This ensures that the computational load is reproducible because OpenTTD is user-input deterministic (R1.b in Section 2). Our experiments use popular and competitive community-developed AIs such as *OtviAI*,[20]

*A summary of the development process of the international community-development process of OpenTTD is available online at https://wiki.openttd.org/FAQ_development.

TABLE 1 *Starting points for OpenTTD experiments:* one saved-game per starting point, with number of OIs ('Road vehicles'), number of artificial intelligence players ('AIs'), and in-game time at the start of each saved game

Input Workload	Road Vehicles	AIs	Game Time, ticks
Save 1	267	4	52 836
Save 2	620	5	93 462
Save 3	1519	7	196 544
Save 4	2685	9	311 688
Save 5	4859	14	542 124
Save 6	6337	14	705 368

the *SimpleAI*, [21] and the *ChooChoo AI*.[22] Both the OtviAI and SimpleAI have been designed to play the game as best as possible using a variety of strategies, while the ChooChoo AI aims by design to build railroad networks that resemble the gameplay of the best players. This gives the generated saved games a good distribution of the types of vehicles built during the game and of gameplay styles, making them representative of typical OpenTTD games played by real players.

4.4 | Measured outputs and main metrics

For each experiment run, the monitoring tool developed in the *Mirror* prototype measures and records a variety of outputs. Only the subset relevant for the experiments in this work is used in the analysis in Section 5. The subset includes the following key in-game metrics on the client-side: the TPS, bandwidth usage in kBps, the number of times the client had to wait for the server ("Number of Waits"), the arithmetic mean of the duration of these waits in milliseconds ("Average Waiting Time"), power consumption in mW, and the MD value.

As the key performance indicator for *performance*, we use the number of TPS the client can simulate; higher values are better. Although measuring the FPS of the game would have been a better metric that gamers could relate to, OpenTTD did not allow for accurate FPS measurements. This is because in the version of OpenTTD used in the experiments, a new frame is only drawn when there are visual changes in the current view of the game world, which can lower the FPS without any relation to the performance of the game. CPU usage also did not turn out to be a good metric to measure performance. This is because if offloading saves CPU time, the game will automatically use the saved CPU time to simulate the next (more) ticks. When the system has to adapt to dynamic conditions, as in the case of policies that offload selectively (in this work, Policies 1–3 in Section 3.3), we also report the *convergence time*, which we define as the time it takes for the system to converge to a stable behavior (measured in real-time seconds or in in-game ticks).

To characterize gameplay experience without requiring costly and time-consuming user studies, we further define two proxy-metrics. We define *game smoothness* as the ability of the game to keep the game wait-free, expressed as a metric as the average time spent waiting from the total time spent in the game session; we also report the "Number of Waits" and "Average Waiting Time" to better analyze game smoothness. We define *game responsiveness* as the time it takes the client to depict updates corresponding to player-input, expressed in in-game time (in ticks) or in real-world time (in milliseconds).

A special type of setup, available only for the Nexus 3 device, was used to measure the effect of offloading on the power consumption on the client-side. The equipment used to do the actual power measuring is the *Power Monitor* by Monsoon Solutions Inc,[23] which,unlike software-only approaches, is a high-frequency high-accuracy device but is also very intrusive (it requires soldering on the mobile hardware).

4.5 | Experiment configuration, runs, and baseline for comparison

To see the effects of using different offloading policies, some of the experiments tested the policies introduced in Section 3.3 for both types of client devices connecting to the t2.medium server. In particular, we want to see how well *Mirror* can enforce the policies and discover what kind of effects they would have on the offloading process. Four policies are defined in Section 3.3, namely *target TPS*, *bandwidth cap*, *offload all*, and *coarse only*. For the *target TPS policy*, the target TPS is set to 20. This is low but still keeps OpenTTD playable and can be enhanced with various client-side rendering-techniques, including frame buffering and interpolation. For the *bandwidth cap policy*, the maximum download rate is set to 5000 B/s and upload rate to 2000 B/s. The policy *coarse-only* will only do coarse-grained offloading. More precisely, the AIs mentioned in Section 4.3 are offloaded. This policy is tested so that we can compare the performance gained from fine-grained offloading to that off coarse-grained. On other experiments, *offload all policy* is used, where we offload all OIs at all times, to provide a binary distinction between offloading and not offloading.

An *experiment run* consists of using a certain client-device, connecting it with one of the server machines through a university Wi-Fi connection, and starting the game at one of the saved games. Each combination was run 3 times with each run lasting around 4 minutes. The power consumption experiments were done by connecting the Nexus 3 with the Amazon EC2.nano server and testing all saved games with and without offloading. Because trial runs on this setup revealed that the power consumption between separate runs with the same parameter combination does not fluctuate significantly, we run each parameter combination only once, for 4 minutes.

An additional set of experiments with *OpenTTD without offloading* were also performed to act as the baseline for comparison. These baseline tests were performed for every combination of client device, and saved game and were also performed 3 times for each combination with a running time of 4 minutes each.

5 | EXPERIMENT RESULTS

This section aims to answer the question of whether the *Mirror* architecture can beneficially perform computation-offloading while keeping the game playable. To this end, we conduct real-world experiments using the experimental setup introduced in the previous section.

Table 2 summarizes the configuration parameters and reported metrics for each experiment. For all the graphs in this section, the horizontal axis contains the number of OIs with larger values corresponding to more computationally demanding workload.

5.1 | Performance

Figure 5 shows, per client device, the TPS results without offloading and with offloading, when using different servers. The Nexus 3 is a fairly old device, and the results show that it cannot manage to run OpenTTD at the maximum TPS even at very low computational loads without offloading. The performance of the game rapidly drops as the computational loads increase. The results show that turning offloading on significantly increases the performance of the game and is able to nearly double the number of ticks that this device can do at higher computational loads. The results for the Nexus 6 device shows a similar trend; although a significantly more powerful device than the Nexus 3, the performance on Nexus 6 also drops significantly at higher computational loads without offloading, while with offloading, it can practically maintain near maximum performance. This shows that, from a pure performance perspective, the *Mirror* architecture is able to beneficially offload very fine-grained functions, making it able to scale well with the computational load of the program.

The results also show that to perform offloading for a single client, a very small server is sufficient to obtain a large (and needed) performance increase at the client-side. This result indicates *Mirror* itself does not add much overhead to the server-side computation, so the server only needs enough computational capacity to run the game faster than the client. This result is expected to hold in practice because, currently, it is not difficult to find servers, desktops, and even laptops that are more powerful than even the most modern mobile device. Moreover, unlike cloud gaming

TABLE 2 *Summary of the experiments:* for each experiment, its section in the article and its focus, its configuration parameters (the resources used for the "Server" and for the "Client", and the "Policy"), and the "Metrics" used to report results

	Focus	Client	Server	Policy	Metrics
§5.1	Performance	Nexus 3, 6	Laptop, EC2 t2.micro, EC2 t2.medium	Offload All (Policy 4)	TPS
§5.2	Game smoothness	Nexus 3, 6	Laptop, EC2 t2.micro, EC2 t2.medium	Offload All	Smoothness, others
§5.3	Game responsiveness	Nexus 3, 6	Laptop, EC2 t2.micro, EC2 t2.medium	Offload All	Delay in ticks, real time
§5.4	Power consumption	Nexus 3	EC2 t2.micro	Offload All, TPS limit (Policy 1)	Power consumption
§5.5	Bandwidth usage	Nexus 3, 6	Laptop, EC2 t2.micro, EC2 t2.medium	Offload All	Download rate
§5.6	Policy vs. performance	Nexus 3, 6	EC2 t2.micro	Policies 1–4	TPS, download rate

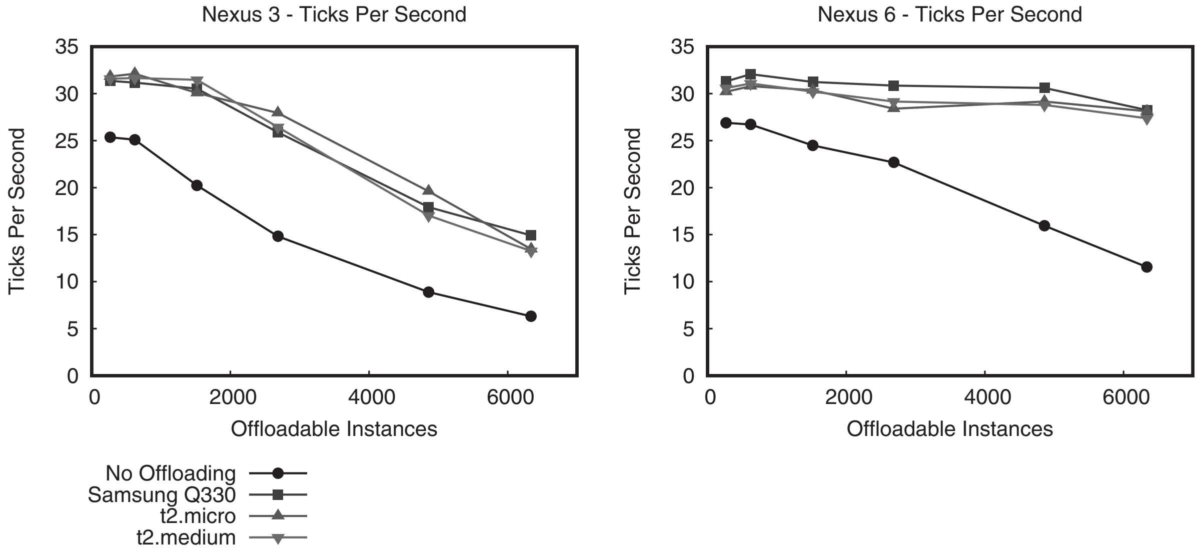

FIGURE 5 *The performance results* in ticks per second

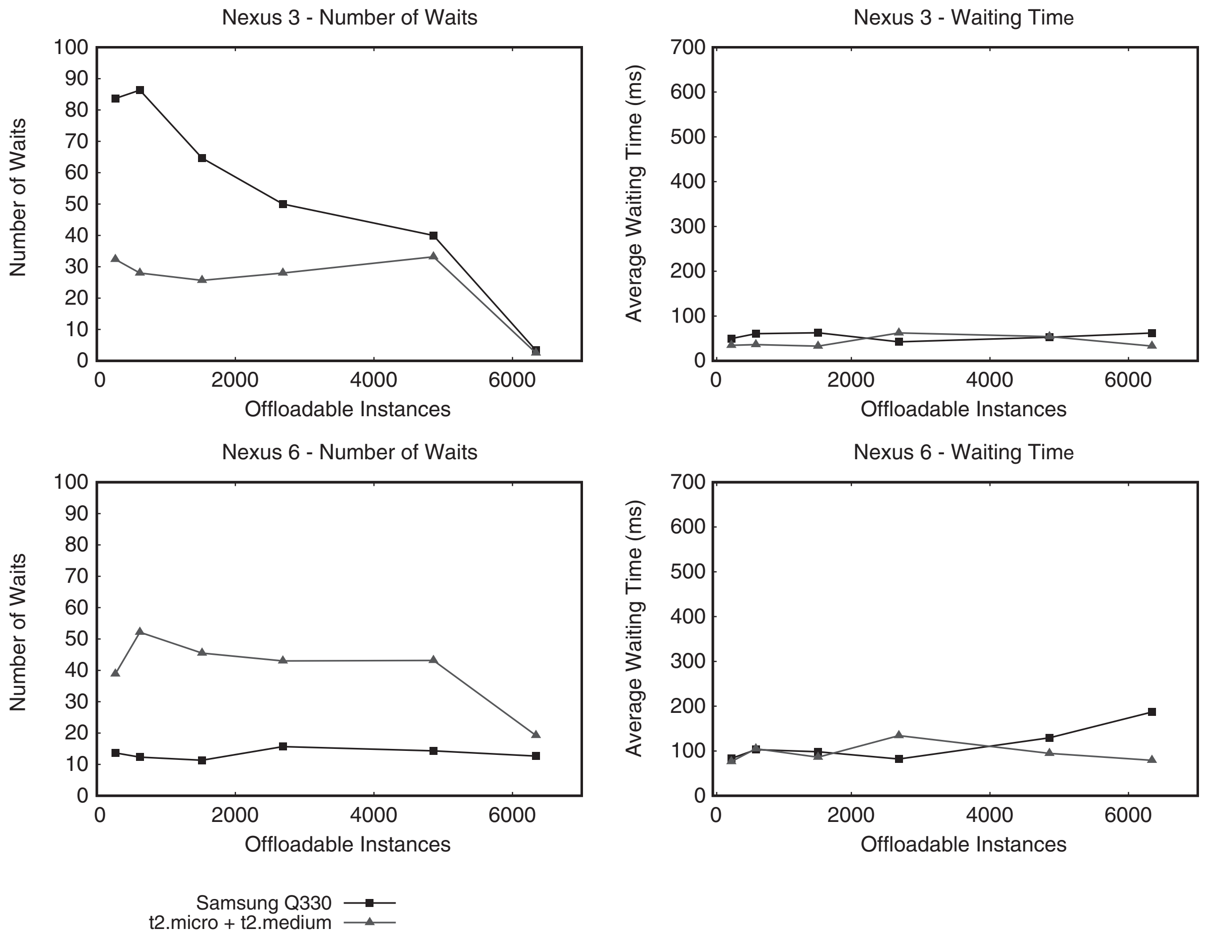

FIGURE 6 *The game smoothness results:* number of waits and the average waiting time as a function of increasing game workload. A "wait" occurs when the protocol in Section 3.4 decides that the client starts to run too fast and thus has to wait. If this happens too, often the player will experience the game as if it is stuttered. The Figures above show that, most of times, there are less than 60 waits during each of the four minute runs, with the average of 40 waits. The waiting time of each wait stays mostly below 100 ms (averaging in 60 ms) even when we increase the number of offloaded entities. While these waits might be noticeable by the user, they are very short and infrequent

(discussed in Section 7), the *Mirror* server is not required to render game world to perform offloading for the client, which further decreases the system requirements for the server-side.

5.2 | Game smoothness

Section 3.4 explained how the *Mirror* architecture synchronizes offloaded events and how this system can potentially cause client-side stuttering. To analyze the severity of such stuttering, we have collected wait-related data from every experiment and summarized the results in Figure 6. Overall, the data we obtained show that the number of waits of each run is low, averaging to about 40 times over the period of four minutes. However, the number also fluctuates much, in particular on the configurations that use cloud-based servers (t2.micro and t2.medium), where the standard deviation (of the number of waits) for each set of the 3 runs can be as high as 40. Most of the times, clients using t2.medium, which is a stronger server, experience less number of waits compared to when t2.micro is used, but this is not always the case. Such fluctuations are likely caused by fluctuating delivery of CPU cycles of the cloud servers as well as fluctuating network condition (the connection between the client and these servers goes through the Internet). Such varying conditions are inherent when we use cloud servers. To average over these variations, in Figure 6, we show the average of t2.micro and t2.medium as representing the performance of using a cloud server and compare it against the performance of using the LAN-connected Q330 server.

The results on the Q330 server show a steadier trend. This server is more consistent in terms of delivered CPU cycles and connection (it is a physical and dedicated server and connected to the client through a LAN). The results of Q330 can thus be seen as the smoothness that we could get if we factor out the above mentioned fluctuation. As a side note, other metrics, namely TPS, power consumption, and bandwidth usage, appear to be less sensitive to such varying conditions. See Figures 5, 7, and 8, which show steady trends despite the use of cloud servers.

From the results, the average waiting time stays consistently at about 60 ms on all servers, which indicates that the extra distance between the Amazon EC2 servers and the local Samsung Q330 server has little influence on this metric. We also analyze the duration of individual waits and find

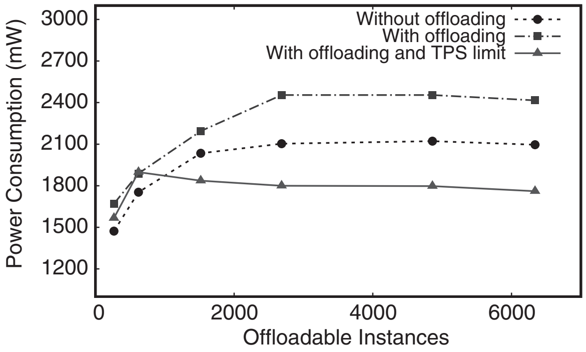

FIGURE 7 *The power consumption results* in milliwatts. Each parameter combination was run once (see Section 4.4)

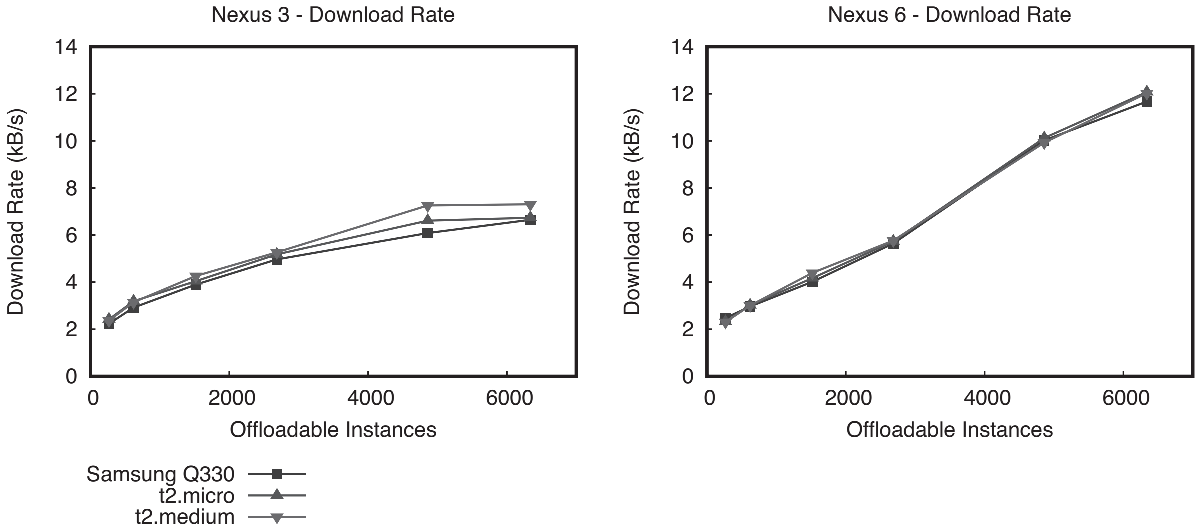

FIGURE 8 *The measured download rate* on the client-side when performing offloading with different servers. The Figures show that the download rate is well below 15 kBps even on the configurations with the highest number of offloaded entities. Even with the older 3G networks, such a rate is very low

that they are generally around 30 ms to 70 ms. Very rarely, they do spike to several hundred milliseconds (the data for individual waits is not depicted in Figure 6), but on the other hand, for example for RTS games, latency of 1000 ms is still considered as acceptable.[24,25]

Interestingly, for both client devices, the number of waits shows a downward trend as the computational load increases. There is a good explanation for this. As the client simulates the game slower, it takes the client longer real-time to get close enough to the server's tick to trigger a wait. This not only gives the server more time to simulate further away from the client but, more importantly, also for the server's tick update messages to arrive at the client even when the network conditions slightly change for the worse.

Overall, with the client having to wait on average around 40 times and for 60 ms each time, the total time spend waiting is less than 1% of the total simulation time. While such waits might be noticeable by the player, they are very short and infrequent. Furthermore, if multiple short waits occur shortly after one another, the player will likely not experience it as stuttering but rather as a slight decrease in performance, which is in general a better experience than stuttering.

5.3 | Game responsiveness

The protocol for user-input synchronization introduced in Section 3.5 incurs delay in response to user inputs directly related to the *MD* value between the client and the server.

The *MD* value measured during the experiments ranged from 1 to 2 ticks, which corresponds to around 30 to 60 ms real-time. The *UF* of the game was set to once per tick, and the *UD* value changed dynamically and was calculated as $3 \cdot MD$. In the worst-case scenario, the client remains 6 ticks behind the server and schedules the user input event 4 ticks in the future of the currently known server tick. This means that the client needs to wait at worst 10 ticks before it can execute the user input event, which corresponds to around 300 ms delay. At best, the client is only one tick behind the server and schedules the event 2 ticks in the future of the server, resulting in a total of 3 ticks wait, which corresponds to around 90 ms delay.

For real-time simulation/strategy (RTS) games such as OpenTTD, we find that *Mirror* leads to delays that typical players would find acceptable.[24,25] However, our results also indicate *Mirror* is not yet capable of supporting all game genres. In particular, *Mirror* is currently unsuitable for fast-paced genres such as e-sports and real-time first-person shooter (FPS) games. For FPS games, players start to notice delays above 100 ms.[26] *Mirror* greatly exceeds this threshold: the worst-case delay is around 300 ms, and the best case scenario, albeit acceptable at 90 ms, is unlikely to happen consistently in practice. The average delay during the experiments was around 150 ms, and for this situation, the client will lag behind also in synchronized content: because the server machine is faster than the client machine, it will likely consistently keep the client *UD* ticks behind it instead of *LD* ticks. For the 150 ms scenario, the *MD* value is equal to 1, and the client 3 stays ticks behind the server. For such games, the response times of *Mirror* are worse than those of existing cloud gaming systems.[27]

5.4 | Power consumption

Figure 7 shows the power consumption on the Nexus 3 mobile device, with and without offloading. (Section 4.4 discusses why the use of the high-frequency, high-fidelity measurement tool means we could not conduct this experiment also on the Nexus 6 device.) The results show a clear trade-off between performance and power consumption.

When using offloading, the power consumption of the client device is nearly 20% higher than without offloading. This could seem surprising because one of the goals of this work is to reduce power consumption through offloading. However, the increased consumption is not spent on offloading. Because offloading has indeed saved local CPU cycles, the OpenTTD simulator uses them to simulate more ticks per second, increasing the performance of the game as witnessed in Figure 5. Thus, the client device not only consumes power to use its Wi-Fi interface to perform offloading but also continues to use its CPU at full capacity. To confirm this explanation, we also conduct power-consumption experiments with the target TPS policy (Policy 1 in Section 3.3) instead of Policy 4. When limiting the number of TPS OpenTTD is allowed to perform to the number of ticks that the Nexus 3 can do without offloading, we see indeed that the client device can save nearly 20% power (curve "with offloading and TPS limit" in Figure 7), outweighing the extra power used by the Wi-Fi connection for the offloading protocol.

These results show a clear trade-off the game developer and the game player have to make, between performance and power consumption. Using offloading on a device that is too "small" to run the game on its own at the maximum speed will result in a significant performance increase but also in increased power consumption. If the device can already run the game at or near the maximum speed, offloading can decrease the power consumption of the game. If the player is satisfied with the performance of the game without offloading, the game should allow the player to limit its simulation speed to save power with offloading.

5.5 | Bandwidth usage

Figure 8 shows the average download rate results on the client-side. The data shows that, even when offloading a very high number of OIs, the download rate is well within the limits of 3G networks, and thus also for the more modern but not everywhere-accessible, 5G, Wi-Fi, and WiMax.

We see that the choice of the server has little impact on the download rate. The data packages the server sent mainly consist of EMs, whereas the messages that are used to control the rest of the offloading process have minimal impact on the download rate as they are often only exchanged once and their number and size do not scale with the number of OIs. The other type of message that does generate some significant download rate on the client-side are tick update messages from the server as these are sent very frequently. The three servers used in the experiments do indeed have different strength. However, the synchronization protocol (Section 3.5) prevents them from running more than UD ticks ahead the client. Consequently, the rate with which they send the EMs and tick update messages will also be quite similar, provided they are all fast enough to be able to outpace the client (they are).

The reason that the download rate is not increasing as fast as the number of OIs for the Nexus 3 is that the device is simulating the game much slower due to the increased computational load caused by those OIs. A slower simulation means less offloaded events happening in the same amount of real-time, which decreases the download rate required. A faster client will therefore require a higher download capacity. As shown in Section 5.1, the Nexus 6 can simulate the game at nearly the maximum speed with offloading even at the maximum computational load. This has caused the download rate of the Nexus 6 device to be much higher than the Nexus 3 device.

The average upload rate was around 200 B/s for all setups, and thus, it is too low to be worth showing. The upload rate consists mainly of tick update messages sent by the client to the server as, again, other types of messages only occur once in a single offloading session. *Mirror* requires very little upload capacity to perform offloading for even high number of instances as an OI does not require the client to send any information to the server except when changing the state of that instance to offloaded or vice-versa. This makes the upload rate scale incredibly well with the number of OIs.

During the experiments, there was no user playing the game, so the upload rate is purely caused by network traffic used by offloading. In a real-life scenario, the required upload rate will be higher as the client needs to send user-input messages to the server. The increase in bandwidth requirements depends on the frequency and size of user inputs for that particular game, but even in sophisticated non-mobile games, this rate is kept under 10 - 20 kBps.[28]

5.6 | Impact of the offloading policy on performance

Figure 9 shows the results of the experiments with all the offloading policies considered in this work. Only the average TPS and the download rate metrics were chosen to compare these policies. This is because the policies are designed to make trade-offs between these two parameters. Ideally, one would also consider the power consumption, but a policy for this could not be implemented as the Nexus 6 device did not have a setup to accurately measure its power consumption. We do show power-consumption results using the Nexus 3 setup but only for Policies 1 and 4 in Section 5.4.

The results of the *target TPS* and the *bandwidth cap* policies both indicate that these policies behave as designed. Both try to maintain just enough offloaded instances to satisfy their conditions and will only offload instances to try to improve their metrics when their limit-conditions are not met. We can actually see in the graphs when these policies start offloading. For example, in the top-left graph in Figure 9, we can see that in the beginning, as the load is increased up to 1500 vehicles, the *target TPS* policy delivers similar TPS as doing no offloading at all (the TPS of *no-offloading* policy). This is to be expected, since so far, doing no offloading still delivers at least 20 TPS, which is the set TPS limit for the *target TPS* policy. When the load is increased further, we can see that the TPS of doing no offloading drops below 20. In contrast, the TPS of the *target TPS* policy increases. This is because the policy becomes active; it senses the dropping TPS and reacts by offloading entities and by doing so manages to keep the TPS around its target 20 TSP. Similarly, in the bottom-right graph, we can see that in the beginning, as the load is increased up to 2700 vehicles, the *bandwidth cap* policy's download rate is about the same as doing no offloading at all. This is because, so far, the download rate is still around or below 5 KBps, which is the cap set for the *bandwidth cap* policy. When the load is further increased, doing no offloading results in increasingly higher download rate, thus breaching the cap. We can see that *bandwidth cap* policy responds correctly as it manages to keep the download rate around 5 KBps (by performing offloading).

The results show that the *target TPS policy* is indeed better at keeping the average ticks per second close to the target, whereas the *bandwidth cap policy* is better at keeping the download rate close to the limit we set. Which of these two should then be used depends on what the player values the most: high TPS or low download-rate. The results show that these two policies are customizable enough for the player to use and to make the trade-off.

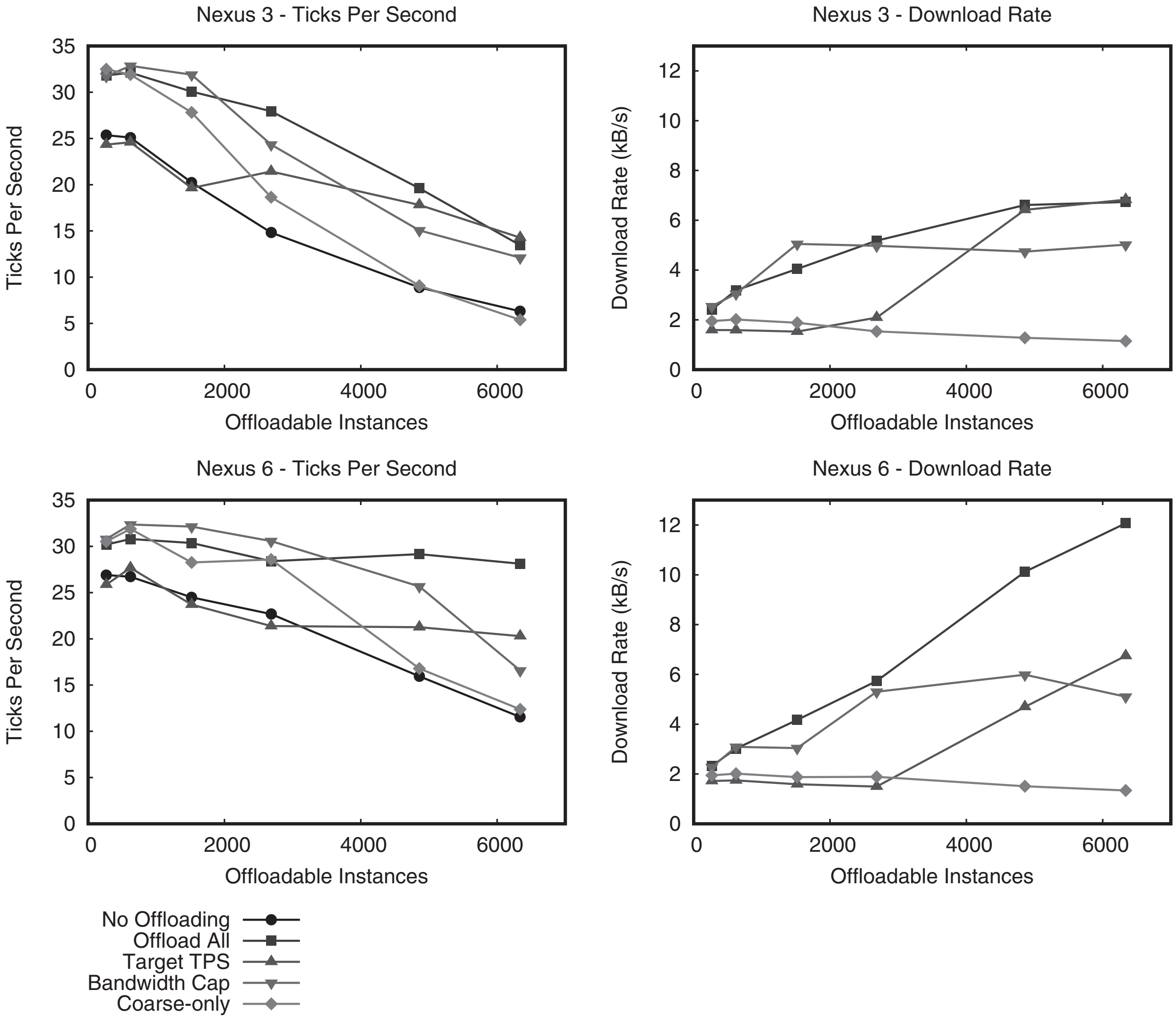

FIGURE 9 The effect on the TPS and download rate on the client-side when using different offloading policies

The results of the *coarse-only policy* (that is, offloading AIs, see Section 4.1) are interesting as they show that, in the early saved games, this policy significantly increases performance, achieving results almost as good as offloading all instances but with lower power-consumption. However, as the number of instances increases, the effects of offloading only the coarse instances drops rapidly, almost to the point that there is barely any difference with not offloading at all. Because this result depends on the ratio of computational load incurred in OpenTTD by AI players (coarse-grained) and road vehicles (fine-grained), we conclude that how well the *coarse-only policy* policy performs will depend on the game code and how the game objects are partitioned.

The time the policies needed to converge to a stable behavior ranged from 30 to 120 seconds depending on the saved game. This *convergence time* could be improved in the future by profiling the game.

6 | DISCUSSION ABOUT LIMITATIONS, THREATS, AND OPPORTUNITIES

We discuss in this section the known limitations of our current *Mirror* architecture and opportunities for future research and identify several threats to the validity of our experiments and opportunities for further experimentation.

6.1 | Known limitations and opportunities for future research

Some important challenges remain to be solved. The first one is how to reduce user input delay to make the architecture usable for a broader range of games, in particular, for games with low tolerance for latency such as the first-person shooter genre. We can do this for example by more accurately measure network latencies. In the current architecture, network delays are measured in ticks instead of milliseconds. This approach was used as it was easier to reason with as nothing was allowed to happen in between ticks. Moreover, how long a tick actually takes in real-time depends on how fast the client can simulate the game. For example, in the experiments, the client ran OpenTTD at 30 TPS tops. There is then around 33 ms real-time in between ticks. Any actual network latency below this would then still be considered to take 33 ms long. By measuring the actual network latency instead, the values for LD and UD can be calculated more accurately, creating tighter bounds on the number of ticks that the client and the server are apart from each other and allowing user input to be executed sooner. A more aggressive way to reduce user input delay is by not letting the client schedule the user input for the server. Instead, the client only notifies the server of a user input, without providing a time stamp. As soon as the server receives the message, it will process the user input in the next tick and immediately sends a message back to the client along with the information on the tick at which this user input was executed by the server (which is running ahead of the client). The client will then process the returned message just like an EM. This avoids the client overestimating how far ahead it needs to schedule the user input and ensures that the user input is always executed at the earliest possible moment. This also eliminates the need to calculate UID.

Recovering from desynchronization is another challenge that we do not address in the current work. Desynchronization happens when the client state, upon entering some in-game time T, turns out to be different than the server state at that time. There are however already plenty of work in the literature addressing similar problems. For example, the problem also occurs in parallel discrete-event simulations (PDES)[29] and in multi-server online games.[30] In PDES, multiple processors are used to simulate individuals. They interact by sending events to each other. The Time Warp algorithm[31] is, for example, an optimistic algorithm used to improve performance. Each processor would optimistically process incoming events without regard of the states and the speed of other processors. It may then happen, that a processor P receives an event sent by another processor that is behind P's simulation time, hence desynchronization has occurred. To recover from desynchronization, each processor saves a *checkpoint*, that is, the last state (its own state) that can be safely inferred to be consistent with that of others. When the processor receives a desynchronizing event, it rolls back its state to the checkpoint, recalculates the correct order of events, and re-execute them. Such a roll back may also cause some events that the processor sent to be incorrect and has to be cancelled. This is done by sending "anti-messages". The same approach can be applied in *Mirror* because *Mirror*'s synchronization algorithm can be seen as an optimistic algorithm. However, forcing the client to do a roll back is not user friendly, and the overhead to facilitate it is expensive for the client (the client needs to regularly save its state). Fortunately, it is sufficient if we only roll back the server to recalculate a new consistent state. If every message in *Mirror* is extended with sent-time time stamp and sequence number, the client and the server will be able to calculate the sequence of events as the client perceives it from the moment of the server's checkpoint. This enables the server to reconstruct a new state that is consistent with the client, and the game can then continue. This can be further improved by employing a *trailing state*. A *trailing state synchronization* algorithm is proposed to synchronize the states of multiple instances of the game running in different server in a multi-server online game.[30] The algorithm avoids Time Wrap's overhead of making checkpoints (which can be substantial if the game has a very large state to save) by deliberately letting some servers to trail behind in their progress. When a server needs to roll back, it then copies the state of a trailing server. This can be employed in *Mirror* by running a second instance of the game in the server to maintain a trailing state.

Although not used in the experiments, the *Mirror* architecture can, at run-time, change the number of offloaded OIs and which type of OIs to offload. In the future, actual profiling data of the game should be fed into the policies, and the architecture should also be able to profile the game while running. This allows the architecture to make better offloading decisions that improves bandwidth and power usage efficiency and provide the developers and players with statistics of the effects of offloading.

6.2 | Threats to validity and opportunities for further experimentation

The currently conducted experiments were only conducted using a fairly stable Wi-Fi connection for the client. Most mobile users will likely use 3G, 4G, or less stable public Wi-Fi connections while traveling. It will be interesting to see whether the game remains playable with these less reliable connections.

In this work, we only implemented *Mirror* for only one mobile game. Experiments with other sophisticated mobile games are needed to discover potential problems of implementation for different games and to measure performance. With regards to game smoothness, however, we believe that performance metric is independent on what kind of game is being played. Instead, it is mainly dependent on the network conditions and the speed at which the client and the server can simulate a single tick, regardless what actually happens during a tick of any game.

Due to the game state at the server-side being representative for the game state at the client-side, it may be possible to offload entire features to the server-side. These features may include features such as logging or the achievement system of games. It would be interesting to see what other things are possible with this system and how implementing this can be made easy for the developers.

The *Mirror* architecture currently only offloads computation. Offloading other types of resources may be very beneficial too, especially graphics. It will be challenging to create a system that can partially offload the rendering process without gaining the negative effects that are inherent to cloud gaming systems.

An automated offline partitioning system will be very useful for developers. However, it is unclear to which extend this can be done without forcing the developers to write the behaviors of their game objects in a pre-defined format, which would defeat the purpose of the automation.

7 | RELATED WORK

In this section, we compare qualitatively our work with prior studies. In contrast to the large body of existing work on offloading, our work focuses on finding out the effects of offloading on *sophisticated mobile games*. For this, we have created a framework that tries to meet the high real-time and user experience requirements of sophisticated mobile games. There are two key aspects that our framework differs from other offloading frameworks. The first one is that we do not use a stateless server but use replication instead. The second one is that the client only needs to subscribe to certain in-game events that it wants to offload. The client then does not need to run the logic of those events itself but only needs to wait for the results that the server sends. These two aspects together makes it possible that the client can continue the gameloop without the need to offload and wait for the result of every call to an offloadable function. Another benefit is that it significantly lowers the overhead required for offloading because the client does not need to serialize game states.

One of the first known proposals for using offloading to alleviate some fundamental constraints in mobile computing was *cyber foraging* by Balan et al,[32] where clients would use local wireless network technologies to discover resourceful machines (*surrogates*). Surrogates are used in two ways. In data staging, a surrogate acts as an Internet cache for a client, predicting and fetching information from distant servers for the client. Alternatively, the client can send code to a surrogate to be executed there. The aforementioned work inspired numerous other projects creating variations of the implementation of cyber foraging.[33-36] They differ from each other primarily in how and which part of applications are profiled and partitioned for local and remote execution, how offloading is decided and controlled, how surrogates are discovered, and how they are managed. Despite these variations, the general idea of computation-offloading remains the same: to reduce the work of the mobile client by using the resources of other machines and using a network connection to transfer the information.

Between 2000 and 2015, as shown by the surveys by Olteanu and Ţăpuş[12] and Khan,[13] offloading has become a fairly popular topic. For example, Flores and Srirama[37] investigated the use of a fuzzy decision engine and evidence-based learning methods to dynamically configure offloading parameters and decisions. Hassan and Chen[38] investigated offloading to other mobile devices instead of to servers. To compensate for weaker computation power, they arrange the devices to form a map-reduce framework, used to offload a map-reduce type of computation. Cai et al[39] proposed an agent-based architecture where the target application is structured as a network of communicating components, each is treated as a mobile agent. These agents are then deployed on a cloud/server. At the runtime, the network is partitioned by migrating some agents to the clients in order to split the computing cost. The partitioning may change dynamically in order to adapt to the changing parameters at the clients and the cloud. Since finding an optimal partition (when given a certain cost function) can be non-trivial, they also investigated the use of genetic algorithm to calculate it. Huerta-Canepa and Dongman[40] investigated computation sharing: if multiple clients can figure out that they need to do the same computation or the same fragment of a computation, only one needs to do it and the result can be shared. Zhang and Jeong[41] investigated the problem of elasticity in offloading and proposed a model that can facilitate it.

There have not been many offloading frameworks that were tested on video games though, and even fewer were created specifically for them. Examples of the latter type are that of Li et al,[9] Chu et al,[10] Kemp et al,[4] and Cuervo et al.[11] However, these works only tested their framework using very simple games such as board games. They also use a stateless server. The client is then required to serialize its game state at every call of an offloaded function and send it to the server along with the function to call. The server will then perform the function and send the game state back. During all this time, the client is required to block the program. This generally works well for simple non-real-time games but is unacceptable for modern games and cannot work for sophisticated real-time strategy (RTS) games such as OpenTTD.

The previously mentioned work by Cai et al[39] showed that an agent-based architecture can handle the offloading of more sophisticated games, eg, one of the games they studied is a prototype game called Skeleton that involves advanced 3D graphics. However, the games studied in the aforementioned work[39] consist of only a handful of agents, implying that the resulting offloading is coarse-grained. In contrast, the OpenTTD game studied in this paper and similarly most other RTS games, generate a large number (thousands) of entities/agents. For such a game, fine-grained offloading as in *Mirror* is a key towards optimizing its offloading.

The work most similar to ours is the work of Kemp et al,[4] which created the Cuckoo framework, a system that can perform computation offload for Android applications. The authors have created a programming model that extends the build process of Android applications. It automatically creates interfaces for Android services that the developer has to implement. At run-time, the Cuckoo framework can decide whether a call to a service should be done locally or remotely. Like many other offloading frameworks, the Cuckoo framework also uses a stateless server and offloads functions on a call by call basis. Another limitation of their work is that their framework enforces the use of the Android build structure and also requires that the server side uses the Java virtual machine.

An existing alternative to offloading for games is *cloud gaming*.[5,42] Cloud gaming attempts to allow users to play the most modern games without the need to constantly upgrade their hardware. Cloud gaming systems execute all of the game's logic on a remote cloud server, render the scene, and send it as either a video or drawing instructions[42] to the client. The client itself is very thin and only acts as a user input terminal and video player. Cloud gaming can be considered the most extreme form of offloading, where only the bare minimum is done on the client, enabling a much thinner client than the ones required for offloading.

Out of the many cloud gaming systems that existed over the years, only a few remain such as StreamMyGame and GamingAnywhere.[5] The exact reason for why commercial cloud gaming systems have not been very successful is outside the scope of this article, but at least from a technical point of view, cloud gaming systems are very sensitive to network latency and package loss,[7,8] both of which are problems inherent to the Internet. Although offloading is influenced by these same problems, it is less sensitive to them. This is because the frequency and the size of data transfer for offloading is much lower compared to the video stream of cloud gaming systems. Moreover, the concept of offloading is more flexible and provides many opportunities to mask network problems. Section 3.4 shows how the *Mirror* framework copes with network jitter. Another advantage that *Mirror* has over cloud gaming is that *Mirror* does not require the server to render the scene as that is still done client-side. This enables the use of very light-weight servers and avoids the use of expensive render farms cloud gaming systems require.

8 | CONCLUSION

The mobile gaming market needs new approaches to cope with the limited computing and power capacity of current mobile devices. Addressing these limitations, in this article, we have proposed the *Mirror* architecture, focusing on computation-offloading for sophisticated, real-life mobile games. *Mirror* works asynchronously, which is necessary in enabling a game to run smoothly. By design, *Mirror* also enables both offline and online offloading decisions, offering the *offloadable entity* as a powerful abstraction. We also design *Mirror* to allow for developer- and/or user-level control of offloading through a policy-based mechanism and equip *Mirror* with four offloading policies. The experimental results obtained using a real-world implementation of *Mirror* and the complex, real-life RTS-game OpenTTD show that computation-offloading can be beneficially used to increase the performance of sophisticated mobile games while keeping it playable. As trade-off, computation-offloading delays user inputs but, as we find experimentally, acceptably so for RTS games. With the current design, the delay can be too long for high-speed action games but is still acceptable for many other game genres, including real-time strategy games. The results also show that offloading may either decrease or increase the power consumption of the mobile device, depending on the performance of the device without offloading and on how the game developer or player appreciate the trade-off between performance and power consumption.

For the future, we plan to address the aspects identified in Section 6: reducing the delay when processing user input to make *Mirror* suitable for fast-paced game genres, exploring dynamic offloading of OIs, conducting experiments with more settings, offloading beyond computation, and automating the software engineering process that underlies offloading.

ACKNOWLEDGMENTS

This work is supported by the NWO/STW Vidi grant MagnaData (14826), the NWO/STW Veni grant AtLarge (11881), and the Dutch projects COMMIT and Commissioner.

ORCID

I. S. W. B. Prasetya http://orcid.org/0000-0002-3421-4635

REFERENCES

1. Newzoo Team. The Global Games Market 2017 Online Market Report. https://newzoo.com/insights/articles/the-global-games-market-will-reach-108-9-billion-in-2017-with-mobile-taking-42/. 2017.
2. Fernando N, Loke SW, Rahayu W. Mobile cloud computing: a survey. *Future Gener Comput Syst*. 2013;29(1):84-106.
3. Kumar K, Liu J, Lu Y-H, Bhargava B. A survey of computation offloading for mobile systems. *Mob Netw Appl*. 2013;18(1):129-140.

4. Kemp R, Palmer N, Kielmann T, Bal H. Cuckoo: A computation offloading framework for smartphones. Paper presented at: International Conference on Mobile Computing, Applications, and Services; 2010; Santa Clara, CA.
5. Huang C-Y, Chen K-T, Chen D-Y, Hsu H-J, Hsu C-H. GamingAnywhere: the first open source cloud gaming system. *ACM Trans Multimed Comput Commun Appl*. 2014;10(1s):10. http://doi.org/10.1145/2537855
6. Chen K-T, Chang Y-C, Tseng P-H, Huang C-Y, Lei C-L. Measuring the latency of cloud gaming systems. Paper presented at: Proceedings of the 19th International Conference on Multimedia; 2011; Scottsdale, AZ.
7. Soliman O, Rezgui A, Soliman H, Manea N. Mobile cloud gaming: Issues and challenges. Paper presented at: International Conference on Mobile Web and Information Systems; 2013; Paphos, Cyprus.
8. Lee Y-T, Chen K-T, Su H-I, Lei C-L. Are all games equally cloud-gaming-friendly? An electromyographic approach. Paper presented at: 11th Annual Workshop on Network and Systems Support for Games; 2012; Venice, Italy.
9. Li Z, Wang C, Xu R. Computation offloading to save energy on handheld devices: A partition scheme. Paper presented at: Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems; 2001; Atlanta, GA.
10. Chu H, Song H, Wong C, Kurakake S, Katagiri M. Roam, a seamless application framework. *J Syst Softw*. 2004;69(3):209-226.
11. Cuervo E, Balasubramanian A, Cho D, et al. MAUI: Making smartphones last longer with code offload. Paper presented at: Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services; 2010; San Francisco, CA.
12. Olteanu A-C, Ţăpuş N. Offloading for mobile devices: a survey. *UPB Sci Bull*. 2014;76(1):1-16.
13. Khan MA. A survey of computation offloading strategies for performance improvement of applications running on mobile devices. *J Netw Comput Appl*. 2015;56:28-40.
14. Steinman JS. Scalable parallel and distributed military simulations using the SPEEDES framework. Paper presented at: Proceedings of the 2nd Electronic Simulation Conference (ELECSIM), Internet; 1995.
15. Shen S, Hu S-Y, Iosup A, Epema DHJ. Area of simulation: mechanism and architecture for multi-avatar virtual environments. *TOMCCAP*. 2015;12(1):8:1-8:24. http://doi.org/10.1145/2764463
16. Jiang MH, Visser OW, Prasetya ISWB, Iosup A. Mirror: A computation-offloading framework for sophisticated mobile games. Paper presented at: IEEE International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM); 2017; Macau, China.
17. Wang X, Chen M, Kim H, Kwon TT, Choi Y, Choi S. Measurement and analysis of online gaming services on mobile wiMAX networks. *Wirel Commun Mob Comput*. 2015;15(7):1198-1211.
18. Shen S, Visser O, Iosup A. RTSenv: An experimental environment for real-time strategy games. Paper presented at: 10th Annual Workshop on Network and Systems Support for Games (NetGames); 2011; Ottawa, Canada.
19. Pylypenko S. OpenTTD Android port free open-source software and documentation. https://sourceforge.net/projects/libsdl-android/files/apk/OpenTTD/. https://wiki.openttd.org/Compiling_and_installing_the_unofficial_Android_port. 2014
20. Visser OW. OtviAI (version 415). www.tt-forums.net/viewtopic.php?t=39707
21. Brumi. SimpleAI v10 - trying to remake the old AI. www.tt-forums.net/viewtopic.php?t=44809
22. Konstapel M. ChooChoo, a train network AI. www.tt-forums.net/viewtopic.php?t=44225
23. Monsoon Power Inc. Power monitor. www.msoon.com
24. Sheldon N, Girard E, Borg S, Claypool M, Agu E. The effect of latency on user performance in Warcraft III. Paper presented at: Proceedings of the 2nd Workshop on Network and System Support for Games; 2003; Redwood City, CA.
25. Raaen K, Grønli T-M. Latency thresholds for usability in games: A survey. Paper presented at: 27th Norsk Informatikkonferanse (NIK); 2014; Halden, Norway.
26. Choy S, Wong B, Simon G, Rosenberg C. The brewing storm in cloud gaming: A measurement study on cloud to end-user latency. Paper presented at: Proceedings of the 11th Annual Workshop on Network and Systems Support for Games; 2012; Venice, Italy.
27. Chen K-T, Chang Y-C, Tseng P-H, Huang C-Y, Lei C-L. Measuring the latency of cloud gaming systems. Paper presented at: Proceedings of the 19th ACM International Conference on Multimedia; 2011; Scottsdale, AZ.
28. Suznjevic M, Matijasevic M. Player behavior and traffic characterization for MMORPGs: a survey. *Multimed Syst*. 2013;19(3):199-220.
29. Fujimoto RM. Parallel discrete event simulation. *Commun ACM*. 1990;33(10):30-53.
30. Cronin E, Filstrup B, Kurc AR, Jamin S. An efficient synchronization mechanism for mirrored game architectures. Paper presented at: Proceedings of the 1st Workshop on Network and System Support for Games; 2002; Braunschweig, Germany.
31. Jefferson DR. Virtual time. *ACM Trans Program Lang Syst*. 1985;7(3):404-425.
32. Balan R, Flinn J, Satyanarayanan M, Sinnamohideen S, Yang H-I. The case for cyber foraging. Paper presented at: Proceedings of the 10th Workshop on ACM SIGOPS European Workshop; 2002; Saint-Emilion, France.
33. Gu X, Nahrstedt K, Messer A, Greenberg I, Milojicic D. Adaptive offloading for pervasive computing. *IEEE Pervasive Comput*. 2004;3(3):66-73.
34. Goyal S, Carter J. A lightweight secure cyber foraging infrastructure for resource-constrained devices. Paper presented at: Proceedings of the 6th IEEE Workshop on Mobile Computing Systems and Applications; 2004; Cumbria, UK.
35. Balan RK, Gergle D, Satyanarayanan M, Herbsleb J. Simplifying cyber foraging for mobile devices. Paper presented at: Proceedings of the 5th International Conference on Mobile Systems, Applications and Services; 2007; San Juan, PR.
36. Yang K, Ou S, Chen H-H. On effective offloading services for resource-constrained mobile devices running heavier mobile internet applications. *IEEE Commun Mag*. 2008;46(1):56-63.
37. Flores H, Srirama S. Adaptive code offloading for mobile cloud applications: Exploiting fuzzy sets and evidence-based learning. Paper presented at: Proceedings of the 4th ACM Workshop on Mobile Cloud Computing and Services; 2013; Taipei, Taiwan.
38. Hassan MA, Chen S. Mobile MapReduce: Minimizing response time of computing intensive mobile applications. Paper presented at: International Conference on Mobile Computing, Applications, and Services; 2011; San Francisco, CA.

39. Cai W, Chan HCB, Wang X, Leung VCM. Cognitive resource optimization for the decomposed cloud gaming platform. *IEEE Trans Circuits Syst Video Technol*. 2015;25(12):2038-2051.
40. Huerta-Canepa G, Lee D. A virtual cloud computing provider for mobile devices. Paper presented at: Proceedings of the 1st ACM Workshop on Mobile Cloud Computing and Services: Social Networks and Beyond; 2010; San Francisco, CA.
41. Zhang X, Jeong S, Kunjithapatham A, Gibbs S. Towards an elastic application model for augmenting computing capabilities of mobile platforms. Paper presented at: International Conference on Mobile Wireless Middleware, Operating Systems, and Applications; 2010; Chicago, IL.
42. Meilander D, Glinka F, Gorlatch S, Lin L, Zhang W, Liao X. Bringing mobile online games to clouds. Paper presented at: IEEE Conference on Computer Communications Workshops; 2014; Toronto, Canada.

How to cite this article: Jiang MH, Visser OW, Prasetya ISWB, Iosup A. A mirroring architecture for sophisticated mobile games using computation-offloading. *Concurrency Computat Pract Exper*. 2018;30:e4494. https://doi.org/10.1002/cpe.4494